



Sandia
National
Laboratories

W. Michael Brown

wmbrown@sandia.gov

Last update: 4/13/10

Geryon

A Unified Coprocessor Library for OpenCL, CUDA-RT, and CUDA Driver

GPGPU Programming

- NVIDIA has most mature compiler options for scientific GPGPU computing and is offering near-future support for ECC
- 3 Low-level APIs for NVIDIA
 - CUDA Runtime API
 - CUDA Driver API
 - OpenCL

CUDA Runtime

- One compiler interface for host and GPU code
 - Host and device code in a single file
 - One executable for host and GPU code
 - Succinct interface
 - Currently the most used API for NVIDIA hardware
 - Compared to OpenCL, can evolve more quickly to efficiently support NVIDIA hardware
-
- Only supports NVIDIA hardware
 - Need to maintain separate code for GPU and CPU calculations
 - Cannot run GPU kernels (*efficiently*) on CPU
 - The compiler interface drives a host compiler and a GPU compiler
 - Very limited host compiler choice (GNU/Visual Studio)
 - Can be limited to certain host compiler versions
 - Must handle flags and error output from both compilers

CUDA Driver

- Freedom in choice and version of host compiler
 - Some, but not much additional flexibility/functionality
 - More similar to the OpenCL API
 - Compared to OpenCL, can evolve more quickly to efficiently support NVIDIA hardware
 - Supports JIT compilation of PTX code for different NVIDIA GPUs
-
- Only supports NVIDIA hardware
 - Need to maintain separate code for GPU and CPU calculations
 - Cannot run GPU kernels (*efficiently*) on CPU
 - Need to store and load GPU kernels in separate file(s) from executable or manage as strings from compiled output inside of code
 - API is much more tedious than CUDA-RT
 - No device emulation (compared to CUDA-RT)
 - CUDA-RT libraries may not be compatible with CUDA Driver

OpenCL

- Supported by many vendors (not restricted to NVIDIA)
 - Can run same kernels on multicore CPU and GPU
 - Driver includes a compiler
 - Device kernels can be compiled on-the-fly for new architectures
 - Freedom in choice of host compilers/versions
 - Smart pointers (reference counting) are nice for library development
-
- New
 - Most drivers are beta and more likely to have bugs
 - No templates, no pointer traversal
 - No device pointer arithmetic on the host
 - Need to store and load GPU kernels in separate file(s) from executable or manage kernels as strings for on-the-fly
 - API is much more tedious than CUDA-RT

OpenCL Differences (1)

for CUDA programmers

- OpenCL does not allow templates
 - Must be done the C-way with preprocessor directives
- OpenCL does not allow pointer traversal
- OpenCL does not allow device pointer arithmetic on the host
 - Must use integer offsets

CUDA: `gpu_sort(vector.begin(), vector.begin()+n);`
OpenCL: `gpu_sort_cl(vector.begin(),n);`

OpenCL Differences (2)

for CUDA programmers

- OpenCL has no 2D memory copy routines for maintaining byte alignments
- In OpenCL, cannot bind existing array to 1D texture/image (to enable cache on certain NVIDIA devices)
- OpenCL has no memset routine for device memory

OpenCL Differences (3)

for CUDA programmers

- Many OpenCL routines require device/context/command_queue objects to be passed around
 - For CUDA, there is an implicit default command queue (stream) and most functions do not require device, context, and queue objects
- OpenCL uses smart pointers
- OpenCL allows for more advanced error handling

Which API?

- CUDA-RT is most convenient for NVIDIA programmer but has annoying compiler restrictions
- CUDA-Driver offers compiler freedom but is more tedious and restricted to NVIDIA
- OpenCL runs on CPU and has significant vendor support but is new with buggy drivers and potentially lagging efficiency for NVIDIA devices

Which API?

- Ideally, would like a consistent interface to all 3 APIs that handles the current problems:
 - There is no overlap in function names for the 3 APIs
 - There is no overlap in types for the 3 APIs
 - Memory management is complicated
 - Host, Host Pinned, Host Pinned Write-Combined, Host Portable, Device, Device 2D with Padding, Constant Device, Texture 2D, Texture 3D, etc.
 - CUDA-RT has 20 memory copy functions requiring up to 8 parameters
 - CUDA-Driver has 17 and OpenCL has 8 requiring up to 11
 - No simple routines for printing device memory for debugging, etc.

Geryon

- Intended to be a simple library for managing all three APIs with a consistent interface
 - Change from one API to another by simply changing the namespace
 - Use multiple APIs in the same code
 - Lightweight (only include files – no build required)
 - Manage device query and selection
 - Simple vector and matrix containers
 - Simple routines for data copy and type casting
 - Simple routines for data I/O
 - Simple classes for managing device timing
 - Simple classes for managing kernel compilation and execution

Geryon

Selecting a low-level API

```
#ifdef USE_CUDART_API
#include "nvc_device.h"           // For device routines
#include "nvc_mat.h"              // For vec/mat containers
#include "nvc_timer.h"             // For timing routines
using namespace ucl_cudart;
#endif

#ifndef USE_CUDADR_API
#include "nvd_device.h"
#include "nvd_mat.h"
#include "nvd_timer.h"
using namespace ucl_cudadr;
#endif

#ifndef USE_OPENCL_API
#include "ocl_device.h"
#include "ocl_mat.h"
#include "ocl_timer.h"
using namespace ucl_opencl;
#endif
```

Geryon

Device Management

```
// Each device object has information on the number of
// devices and device properties
// - See docs for full list of device property accessors
UCL_Device dev;
if (dev.num_devices()==0) exit(1);
cout << dev.name(0) << " has " << dev.gigabytes(0)
    << " of memory.\n";

// You must have one device object for each device you
// will be using. For CUDA each device object must be on
// a separate thread or process. The set() command will
// initialize the chosen device for use.
dev.set(0);

// The set command creates a default command queue that
// is stored in the device object (in CUDA this is the
// null stream)
command_queue &cq=dev.cq();

// You can add/delete additional command queues (streams)
dev.push_command_queue();
command_queue &cq_kernels=dev.cq(1);
dev.pop_command_queue();
```

```
// To block until all commands in the default queue
// (stream) have completed (synchronization)
dev.sync();

// To block until all commands in the second command
// queue associated with a device have completed:
dev.sync(1);
// -- or --
ucl_sync(dev.cq(1));
```

Geryon

Vector and Matrix Containers

Geryon Vector/Matrix

- Containers for host, device, and texture
 - `UCL_H_Vec`, `UCL_H_Mat`, `UCL_D_Vec`, `UCL_D_Mat`,
`UCL_Image`
- In addition to the usual conveniences, the separate type for matrices allows the library to maintain the correct byte-alignment for each row (efficient global memory access)
 - The library is lenient and allows copies from vector to matrix types and vice versa

Geryon Vector/Matrix

- Functions operating on containers make use of templates and enumerated traits
 - Almost all branches in library can be eliminated at compile time
 - Most routines require no additional overhead
- Currently all containers are row-major

Geryon Vector/Matrix

- When allocating memory, a device object is passed
 - Know where to allocate the memory
 - The container stores a smart pointer to the default command queue for the device
 - Unless a command queue is explicitly passed for copy, print, zero, etc. routines, the default command queue is used

Geryon – Host Allocation

```
// Allocate page-locked memory (2 rows, 3 columns)
UCL_H_Mat<float> h_mat;
h_mat.alloc(2,3,dev);
// --- or ---
UCL_H_Mat<float> h_mat(2,3,dev);

// If you want to explicitly specify the allocation type:
// - set kind=UCL_RW_OPTIMIZED for page-locked
// - set kind=UCL_WRITE_OPTIMIZED for write-combined
// - set kind=UCL_NOT_PINNED for standard malloc
h_mat.alloc(2,3,dev,kind);
// --- or ---
UCL_H_Mat<float> h_mat(2,3,dev,kind);

// Allocate page-locked vector with 6 elements
UCL_H_Vec<float> h_vec(6,dev);
// --- or ---
h_vec.alloc(6,dev);
// ...
```

Geryon – Device Allocation

```
// Allocate device memory (2 rows, 3 columns)
UCL_D_Mat<float> d_mat;
d_mat.alloc(2,3,dev);
// --- or ---
UCL_D_Mat<float> d_mat(2,3,dev);

// You can specify that the memory will be read- or
// write-only within a device kernel
// -kind=UCL_READ_ONLY, UCL_WRITE_ONLY, or UCL_READ_WRITE
d_mat.alloc(2,3,dev,kind);
// --- or ---
UCL_D_Mat<float> d_mat(2,3,dev,kind);

// Allocate device vector with 6 elements
UCL_D_Vec<float> d_vec(6,dev);
// --- or ---
d_vec.alloc(6,dev);
// ...
```

Geryon - Views

- The containers can also use existing memory allocations on the host and device instead of allocating new memory
 - When the container is “viewing” another allocation, it will not free the memory allocations

Geryon – Views

```
// Allow access to an existing Geryon memory allocation
mat.view(existing_mat);
// Allow access to first n elements of Geryon allocation
mat.view(existing_mat,n);
// Allow access to slice of Geryon allocation
mat.view(existing_mat,rows,cols);

// Allow access to an existing CUDA or OpenCL allocation
// - For double, ptr is double*, CUdeviceptr, or cl_mem
// - n is number of elements (not bytes)
// - For views of existing device memory, restricted
//   to API for that pointer (CUdeviceptr, cl_mem, etc)
// - Must pass a device (for default command queue) or
//   an specific command_queue (cq)
mat.view(ptr,n,dev);           // -- or --
mat.view(ptr,n,cq);           // -- or --
mat.view(ptr,rows,cols,dev);    // -- or --
mat.view(ptr,rows,cols,cq);
```

Geryon – Views

```
// A view may start at a specified offset (in elements
// not bytes) from the beginning of an existing
// allocation
mat.view_offset(offset,existing_mat);           // -or-
mat.view_offset(offset,existing_mat,n);          // -or-
mat.view_offset(offset,existing_mat,rows,cols);  // -or-
mat.view_offset(ptr,n,dev);                     // -or-
mat.view_offset(ptr,n,cq);                      // -or-
mat.view_offset(ptr,rows,cols,dev);              // -or-
mat.view_offset(ptr,rows,cols,cq);

// When “viewing” Geryon containers, use the row_size()
// function to determine the offset of device matrices
mat.view_offset(mat.row_bytes()*nrows,mat);
```

Geryon – Zero and Free

```
// You can set every element to zero on host or device
// -- This is currently not optimized for OpenCL and
//    should not be used in a loop
h_mat.zero();
d_mat.zero();

// To zero the first n elements (not bytes)
h_mat.zero(n);
d_mat.zero(n);

// Free host or device memory
// - Memory is automatically freed on destruction
// - If alloc() is called, any previous memory is freed
h_mat.clear();
d_mat.clear();
```

Geryon – Command Queues

```
// You can access the default command queue (stream)
// associated with a container
command_queue &cq=h_mat.cq();

// If you do not have access to a device object for
// memory allocation, you can use another matrix or
// vector object to allocate the memory on the same
// device with the same command queue
h_mat.alloc(2,3,h_vec);

// The command queue executes asynchronously on the
// device while host code is running
// You can block until the default command queue
// associated with container is done with all commands.
h_mat.sync();
```

Geryon – Host element access

```
// You can use 1D and 2D indexing on both vector and
// matrix types
cout << "The 4th element is: " << h_mat(3) << endl;
// --- or ---
cout << "The 4th element is: " << h_mat(1,0) << endl;

// Setting the 4th element to zero
h_mat(3)=0;
// --- or ---
h_mat(1,0)=0;

// Get the pointer to the beginning of the array
float *h_begin=h_mat.begin();
// Get the pointer to one past the last element
float *h_end=h_mat.end();
```

Geryon – Device element access

```
// Currently device element access is limited to kernels  
// run on the device. Direct element accessing for  
// debugging purposes might be added in future versions  
  
// To access the API type that should be passed to a  
// kernel for array access use the begin() member  
add_kernel_param(&d_mat.begin());
```

Geryon – Vector/Matrix Size

```
// The following accessors work for all vector/matrix and
// host/device containers:
cout << "The matrix has " << h_mat.numel()
    << "elements, " << h_mat.rows() << " rows, and "
    << h_mat.cols() << " columns.\n";

// Because some containers use extra columns for padding
// the row_size() and row_bytes() accessors can be used
// to determine the stride for a row in elements and
// bytes respectively.
cout << "The number of columns is: " << h_mat.cols()
    << " and the number of columns with padding is: "
    << h_mat.row_size() << endl;

// The pointer for row i and column j is:
h_mat.begin()+h_mat.row_size()*i+j
```

Geryon

Memory Copy Routines

Geryon – ucl_copy

- ucl_copy is the copy routine for all Geryon containers
- The function is passed an *async* argument for any copy
 - if (*async*==false) perform blocking copy
 - if (*async*==true) perform asynchronous copy using default command queue on destination container
 - if (*async*==command_queue) perform asynchronous copy using the specified command_queue
- The argument is ignored for some copies (e.g. host->host copy cannot be asynchronous)

Geryon – ucl_copy

- If no size parameter is passed to ucl_copy, the entire source container is passed
- If 1 size parameter is passed to ucl_copy, this parameter is interpreted as a number of elements
- If 2 size parameters are passed, this parameter is interpreted as the number of rows and columns
- The routine understands and handles padding for device matrices correctly

Examples (page 1)

(x's represent alignment padding - to maintain alignment)

(o's represent a larger matrix in memory)

(vectors represented as single row)

| <i>dst</i> | <i>src</i> | <i>command</i> |
|---|--|--|
| $\begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,async)</code> |
| $\begin{matrix} 0 & 1 & 2 & 3 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,4,async)</code> |
| $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,async)</code> |
| $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,async)</code> |
| $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,async)</code> |
| $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,6,async)</code> |
| $\begin{matrix} 0 & 1 & 2 \\ 4 & 5 & 6 \\ 8 & 9 & 10 & 11 \end{matrix}$ | $\begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{matrix} \leftarrow$ | <code>ucl_copy(dst,src,2,3,async)</code> |

Examples (page 2)

*(x's represent alignment padding - to maintain alignment)
(o's represent a larger matrix in memory)
(vectors represented as single row)*

| <i>dst</i> | <i>src</i> | <i>command</i> |
|---|--------------|--|
| $\begin{matrix} 0 & 1 & 2 & x & x \end{matrix}$ | \leftarrow | $\begin{matrix} 0 & 1 & 2 \end{matrix}$ |
| $\begin{matrix} 3 & 4 & 5 & x & x \end{matrix}$ | | <code>ucl_copy(dst,src,async)</code> |
| $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix}$ | \leftarrow | $\begin{matrix} 0 & 1 & 2 & x & x \\ 3 & 4 & 5 & x & x \end{matrix}$ |
| $\begin{matrix} 0 & 1 & 2 & o & o \\ 3 & 4 & 5 & o & o \\ 0 & 0 & 0 & o & o \end{matrix}$ | \leftarrow | $\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 0 & 0 & 0 \end{matrix}$ |
| $\begin{matrix} 0 & 1 & 2 & o & o \\ 3 & 4 & 5 & o & o \\ 0 & 0 & 0 & o & o \end{matrix}$ | \leftarrow | <code>ucl_copy(dst,src,2,3,async)</code> |
| $\begin{matrix} 0 & 1 & o & o & o \\ 2 & 3 & o & o & o \\ 0 & 0 & o & o & o \end{matrix}$ | \leftarrow | <code>ucl_copy(dst,src,2,2,async)</code> |

Examples (page 3)

(x's represent alignment padding - to maintain alignment)

(o's represent a larger matrix in memory)

(vectors represented as single row)

| <i>dst</i> | <i>src</i> | <i>command</i> |
|--------------------|-------------------------|------------------------------------|
| <i>0 1 2 0 0</i> | <i><-- 0 1 2 3 4</i> | <i>ucl_copy(dst,src,2,3,async)</i> |
| <i>5 6 7 0 0</i> | <i>5 6 7 8 9</i> | |
| <i>0 0 0 0 0</i> | <i>10 11 12 13 14</i> | |
| <i>0 1 2 5 6 7</i> | <i><-- 0 1 2 3 4</i> | <i>ucl_copy(dst,src,2,3,async)</i> |
| | <i>5 6 7 8 9</i> | |
| | <i>10 11 12 13 14</i> | |

Geryon – ucl_copy

- If it is desired to copy using a non-zero offset from the beginning of a source or destination allocation, use *views*
- If it is desired to copy to a source or destination allocation that was not performed within Geryon, use *views*

Geryon – type casting

- Type casting is automatically performed by `ucl_copy` when the source and destination matrices differ
- When type casting is performed, a cast buffer is allocated
 - For casting within a loop, it is more efficient to allocate a cast buffer only once and pass this to the `ucl_cast_copy` routine

Geryon

I/O

```
// Print the entire vector matrix to standard out
cout << d_mat << endl;
// --- or ---
ucl_print(d_mat);
// Print first n elements of vector or matrix
ucl_print(d_mat,n);
// Print first n elements of vector to stderr
ucl_print(d_mat,n,cerr);
// Print first n elements of vector with comma delim
ucl_print(d_mat,n,cerr,",,");

// Print upper 2x3 slice of matrix
ucl_print(d_mat,2,3);
// Print upper slice with , delimiter and ; at row end
ucl_print(d_mat,2,3,cout,",",",",";\n");

// !!! For const qualified containers, a device object
// !!! must be passed as the last argument of ucl_print
ucl_print(d_mat,2,3,dev);
```

Geryon

Device Timing

```
UCL_Timer timer;

// Initialize a timer for use on the default queue
timer.init(dev);
// --- or --- initialize for specific command queue
timer.init(dev,dev.cq());

// Time all commands added to command queue after start()
// and before stop()
timer.start();
// ...
timer.stop();

// Block on the host until all commands in the command
// queue up to stop() have completed and return time
double milliseconds=timer.timer();
// --- or ---
double seconds=timer.seconds();

// For multiple timings, init() need only be called once
// For multiple timings, timer() or seconds() must be
// called before issuing another start() command
```

Geryon

Vector Add Example

```
#include "nvc_device.h"
#include "nvc_mat.h"
#include "nvc_timer.h"
using namespace std;
using namespace ucl_cudart;

int main() {
    UCL_Device dev;
    if (dev.num_devices()==0) exit(1);
    dev.set(0);

    UCL_H_Vec<double> a(6,dev,UCL_WRITE_OPTIMIZED), b(6,dev,UCL_WRITE_OPTIMIZED);
    UCL_D_Vec<float> dev_a(6,dev,UCL_READ_ONLY), dev_b(6,dev,UCL_READ_ONLY);
    UCL_D_Vec<float> answer(6,dev,UCL_WRITE_ONLY);
    UCL_Timertimer_com(dev), timer_kernel(dev);

    for (int i=0; i<6; i++) { a[i]=i; b[i]=i; }

    timer_com.start();
    ucl_copy(dev_a,a,true);
    ucl_copy(dev_b,b,true);
    timer_com.stop();

    timer_kernel.start();
    // Call kernel here with dev_a.begin(), dev_b.begin(), answer.begin()
    timer_kernel.stop();

    cout << "Answer: " << answer << endl << "Input copy time: "
        << timer_com.seconds() << endl << "Kernel time: "
        << timer_kernel.seconds() << endl;

    return 0;
}
```

Geryon

Kernel Stuff

(for OpenCL and CUDA Driver)

Geryon – Program/Kernel

- A *program* in Geryon is 1 or more kernel functions contained in a single file or string
 - Program objects are for loading and compiling kernels
 - OpenCL can compile from source, CUDA must use PTX or binary
- A *kernel* in Geryon is a single kernel from a program
 - Kernel objects are for running kernels on a device

```
// Program takes context & default command queue from dev
UCL_Program program(dev);

// Read a program from file and compile
// - Flags are currently ignored for CUDA
// - If flags=="BINARY", no compilation is performed
string flags="-cl-fast-relaxed-math -D Scalar=float";
program.load("foo.cl",flags.c_str());
// --- or --- read program from const char * string
program.load_string(foo_str,flags.c_str());
// --- or --- load a precompiled binary (currently CUDA)
program.load_binary(filename);

// To get a build log associated with compilation
string clog;
program.load("foo.cl",flags.c_str(),&clog);
// --- or ---
program.load_string(foo_str,flags.c_str(),&clog);

// Get a kernel object that uses the vec_add function
// in the program.
// - Takes default command queue from the program
UCL_Kernel k_vec_add(program,"vec_add")
```

```
// Set function arguments to vec_add
k_vec_add.set_arg(0,&dev_a.begin());
k_vec_add.set_arg(1,&dev_b.begin());
k_vec_add.set_arg(2,&answer.begin());
// --- or --- add 1,2,3, or 4 args at a time
k_vec_add.add_args(&dev_a.begin(),&dev_b.begin(),
&answer.begin());

// Set up 1-dimensional execution grid to add 6 elements
size_t num_blocks=2, block_size=3;
k_vec_add.set_size(num_blocks,block_size);

// Enqueue the kernel in the default command queue
k_vec_add.run();
```

```
// To set up a 2-dimensional kernel execution grid
k_vec_add.set_size(num_blocks_x,num_blocks_y,
                   block_size_x,block_size_y);
// or to set up a 2-dimensional kernel execution grid
// with 3-D blocks
k_vec_add.set_size(num_blocks_x,num_blocks_y,
                   block_size_x,block_size_y,block_size_z);

// To run in a command queue other than the default
command_queue &vec_add_q=dev.cq(1);
k_vec_add.run(vec_add_q);
```

Geryon

Writing Kernels that are OpenCL and CUDA
Compatible

Must write your own preprocessor definitions

- Example: define NV_KERNEL only when compiling kernel with CUDA... (next page)

```
#ifdef NV_KERNEL

#define DEV_PTR Scalar
#define GLOBAL_ID_X threadIdx.x+INT_MUL(blockIdx.x,blockDim.x)
#define GLOBAL_ID_Y threadIdx.y+INT_MUL(blockIdx.y,blockDim.y)
#define THREAD_ID_X threadIdx.x
#define THREAD_ID_Y threadIdx.y
#define BLOCK_ID_X blockIdx.x
#define BLOCK_ID_Y blockIdx.y
#define BLOCK_SIZE_X blockDim.x
#define BLOCK_SIZE_Y blockDim.y
#define __kernel extern "C" __global__
#define __local __shared__
#define mul24 __mul24

#else

#define DEV_PTR __global Scalar
#define GLOBAL_ID_X get_global_id(0)
#define GLOBAL_ID_Y get_global_id(1)
#define THREAD_ID_X get_local_id(0)
#define THREAD_ID_Y get_local_id(1)
#define BLOCK_ID_X get_group_id(0)
#define BLOCK_ID_Y get_group_id(1)
#define BLOCK_SIZE_X get_local_size(0)
#define BLOCK_SIZE_Y get_local_size(1)
#define __syncthreads() barrier(CLK_LOCAL_MEM_FENCE)

#endif
```

No templates in OpenCL

- Compile kernel with flags that define template arguments “`-D Scalar=float -D Ordinal=int`” instead

```
#ifdef NV_KERNEL
template <class Scalar, class Ordinal>
#endif
__kernel void vec_add(DEV_PTR *a, DEV_PTR *b, DEV_PTR *ans) {
    Ordinal i=GLOBAL_ID_X;
    ans[i]=a[i]+b[i];
}
```

- The string for a given data type can obtained using `ucl_template_name<>()`
 - `string flag=string("-D Scalar=")+ucl_template_name<Scalar>();`

Geryon

Running OpenCL Kernels on CPUs

Don't need device allocations or host-device copies

- Use views:

```
UCL_H_Vec host_mem;
host_mem.alloc(6,dev);
UCL_D_Vec device_mem;
// Only allocate device memory if the device is not a CPU
if (dev.device_type==UCL_CPU)
    device_mem.view(host_mem);
else
    device_mem.alloc(6,dev);
```

- `ucl_copy` will do nothing if the pointer/offset for the source and destination are the same

Geryon

Error Handling

Geryon – CUDA and OpenCL Errors

- No error checking for out-of-bounds indexing
- Most CUDA and OpenCL errors are handled with output to stderr and exit.
 - Most of these should be handled by the software developer outside of Geryon
- The current exceptions are kernel compilation and memory allocation
 - For these functions, error flags and/or build logs are returned that can be handled by the developer
 - Preprocessor flags must be set if the developer does not want Geryon to handle the errors

Geryon – Preprocessor Directives

- Define
 - `UCL_DEBUG` – turn on some sanity and mat/vec size checks (mostly assert statements)
 - `UCL_NO_EXIT` – do not exit with a message to stderr when an error occurs in kernel compilation or memory allocation
 - `UCL_NO_API_CHECK` – turn off error checking for CUDA and OpenCL API calls that do not involve memory allocation or kernel compilation (*unlikely to improve performance*)

```
// To check for memory allocation errors
// -- Define UCL_NO_EXIT preprocessor directive
if (d_mat.alloc(6,dev)!=UCL_SUCCESS) {
    cerr << "Could not allocate 6 elements on device "
        << dev.name() << endl;
}

// To check for compiling errors
// -- Define UCL_NO_EXIT preprocessor directive
string clog;
int err=program.load("test.ocl","","",&clog);
if (err!=UCL_SUCCESS) {
    if (err==UCL_FILE_NOT_FOUND)
        cerr << "Could not find test.ocl\n";
    if (err==UCL_COMPILE_ERROR)
        cerr << "Problem compiling test.ocl:\n"
            << clog << endl;
    exit(1);
}
if (kernel.set_function(program,"foo")!=UCL_SUCCESS)
    cerr << "Could not find function foo in test.ocl\n";
```

Geryon

Current Status and Future Work

Current Status

- Texture/Image Containers are not complete
- Library tests are in place for common use, but not all routines/combinations are covered
- zero() routine for OpenCL is inefficient
- Efficiency of 2D copy in OpenCL (performed as a series of 1D copies has not been tested)
- Need to test most efficient type cast procedure (host or device cast)
 - Currently type casting is performed on the host
 - For archs that support double precision, other options

To do

- Column major transpose copy